Chapter 3

Assembly Language: Part 1

Machine language program (in hex notation) from Chapter 2

FIGURE 3.1

0004	Load ac from location 4
2005	Add to ac from location 5
1006	Store result in location 6
FFFF	Halt
000F	First number
0001	Second number
0000	Result field

Symbolic instructions

- To make machine instructions more readable (to humans), let's substitute mnemonics for opcodes, and decimal numbers for binary addresses and constants.
- The resulting instructions are called *assembly language instructions.*

Machine language instruction:
0000 0000000000100

Assembly language instruction:
ld   4

A *directive* directs us to do something. For example, the define word (dw) directive tells us to interpret the number that follows it as a memory data word.

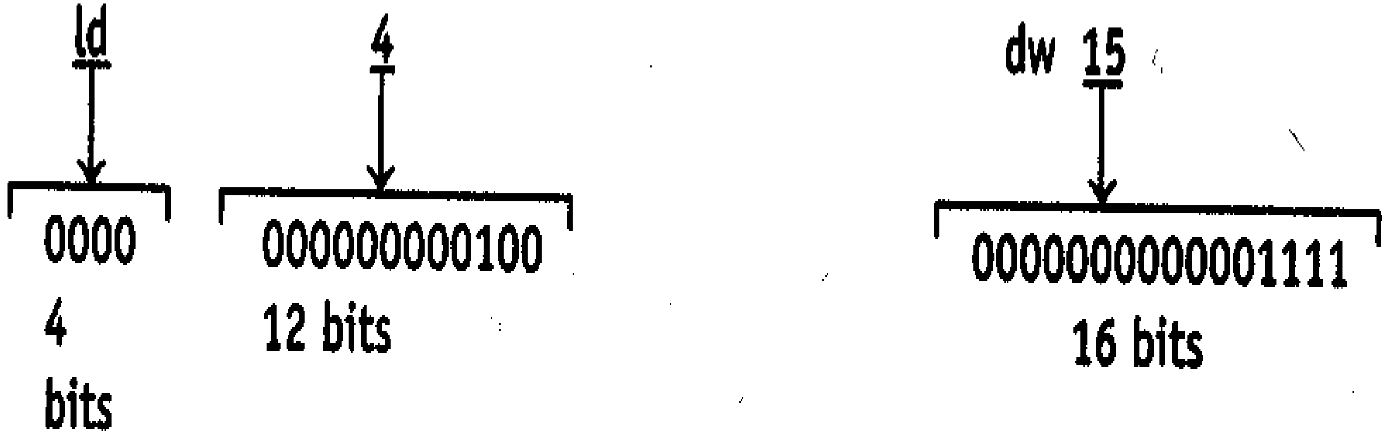
Defining data with dw

Let’s represent data using the dw (define word) directive. For example, to define the data word 0000000000001111 (15 decimal), use

```
dw      15
```

The ld mnemonic versus the dw directive

FIGURE 3.2



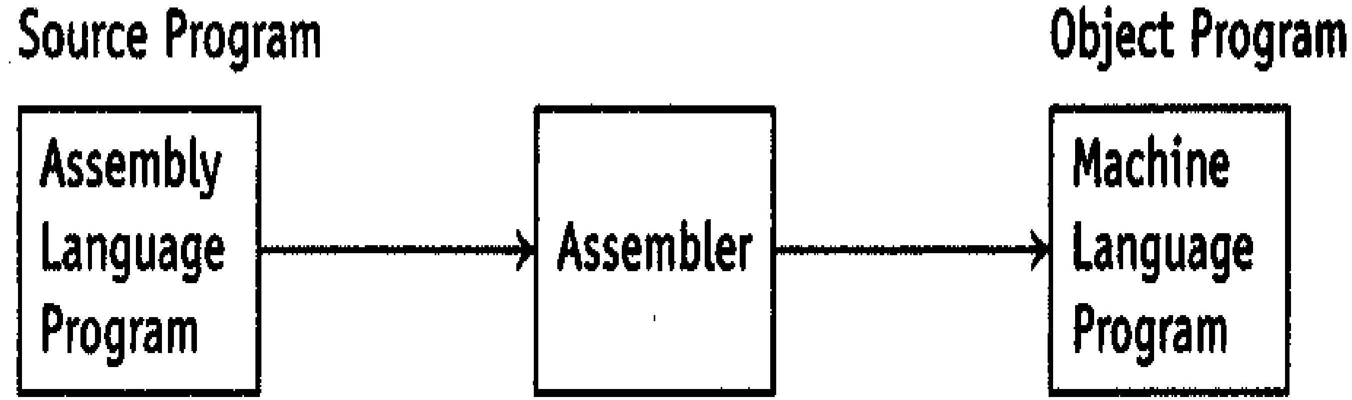
Assembly language—a symbolic representation of machine language

FIGURE 3.3

```
ld    4
add   5
st    6
halt
dw    15
dw    1
dw    0
```

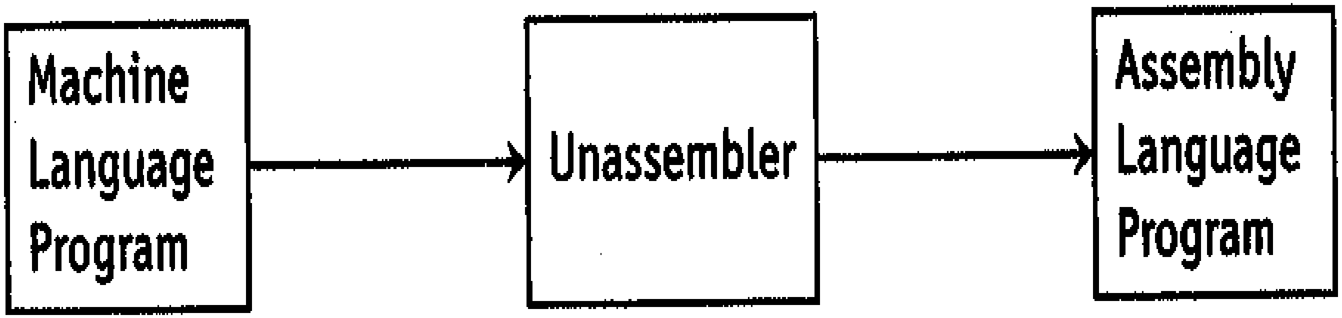
The CPU cannot understand assembly language

FIGURE 3.4



Unassembler does the reverse of an assembler

FIGURE 3.5



Commenting is important

FIGURE 3.6

```
; assembly language program that adds two numbers

        ld    4      ; get first number      ←address 0
        add   5      ; add second number
        st    6      ; store sum in memory
        halt         ; return to OS          ←address 3

;       data area
        dw    15     ; first number          ←address 4
        dw    1      ; second number
        dw    0      ; store sum here        ←address 6
```

Let’s modify previous program to add three numbers

- Requires the insertion of a 2nd add instruction.
- The insertion changes the addresses of all the items that follow it.
- This change of addresses necessitates more changes, resulting in a clerical nightmare.
- Solution: use labels

FIGURE 3.7

```
; assembly language program that adds three numbers

        ld   5        ; get first number
        add  6        ; add second number
        add  8        ; add third number
        st   7        ; store sum in memory
        halt          ; return to OS

;       data area
        dw   15       ; first number          ←address 5
        dw   1        ; second number         ←address 6
        dw   0        ; store result here     ←address 7
        dw   4        ; third number          ←address 8
```

If we use labels instead of number addresses, insertions into an assembly language program don’t cause problems.

A *label* is a symbolic address.

Use labels

FIGURE 3.8

```
; assembly language program that adds two numbers

          ld    n1      ; get first number
          add   n2      ; add second number
          st    result  ; store sum in memory
          halt          ; return to operating system

;         data area
n1:       dw    15      ; first number
n2:       dw    1       ; second number
result:   dw    0       ; store sum here
```

Use labels

FIGURE 3.9 ; assembly language program that adds three numbers

```
        ld   n1      ; get first number
        add  n2      ; add second number
        add  n3      ; add third number    ←insertion
        st   result  ; store sum in memory
        halt         ; return to operating system

;       data area
n1:     dw   15      ; first number
n2:     dw   1       ; second number
n3:     dw   4       ; third number        ←insertion
result: dw   0       ; store sum here
```

Absolute versus symbolic addresses

ld 4

4 is an *absolute address.*

ld x

x is a *symbolic address.*

Good formatting

Improves the readability (by humans) of an assembly language program

FIGURE 3.10 a)

```
start:ld     n1           ; A properly formatted program
st   x  ; is easy to read
       halt
  n1  : dw     5
    x:dw 0; receives copy of n1
```

b)

```
start:   ld    n1        ; A properly formatted program
         st    x         ; is easy to read
         halt
n1:      dw    5
x:       dw    0         ; receives copy of n1
```

Comments

Addresses, data

Mnemonics, directives

Labels

It is ok to put multiple labels on a single item

FIGURE 3.11

```
        ld   x
        ld   y
        ld   z
        halt
x:
y:
z:
        dw   5
```

Action of an assembler

- Replaces mnemonics with opcodes.
- Replaces symbolic addresses with absolute addresses (in binary).
- Replaces decimal or hex absolute addresses with binary equivalents.

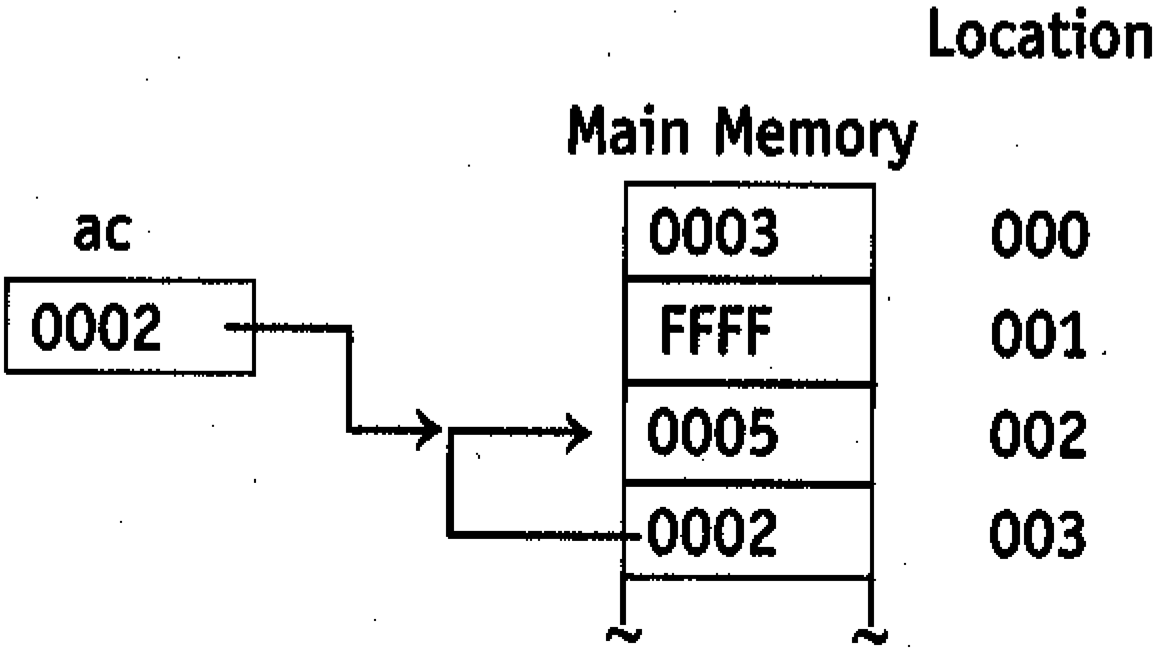
A label to the right of a dw represents a pointer

FIGURE 3.12

Location	Object Code		Source Code		
0	0003		ld	p	; make ac reg point to n1
1	FFFF		halt		
2	0005	n1:	dw	5	
3	0002	p:	dw	n1	; this word points to n1

Right after ld instruction executed

FIGURE 3.13



mas assembler

- Translates a “.mas” file (assembly language) to a “.mac” file (machine language).
- For example, when **mas** translates fig0308.mas, it creates a file fig0308.mac containing the corresponding machine language program.
- **mas** also creates a listing file fig0308.lst.

Assume we have an assembly language program in a file named fig0308.mas.

The next slide shows you how to assemble it using the **mas** assembler.

Using the mas assembler

FIGURE 3.14

```
C:\H1>mas
Machine-Level Assembler Version x.x
Enter input file name and/or args, or hit ENTER to quit.
fig0308
Input file  = fig0308.mas
Output file = fig0308.mac
List file   = fig0308.lst
```

You can also enter the file name on the command line (then sim will not prompt for one):

```
mas fig0308
```

We get a “.mac” file (machine language) when we assemble an assembly language program. We can then run the “.mac” file on **sim**. The next slide shows how to use **sim** to run the “.mac” file fig0308.mac.

FIGURE 3.15

```
C:\H1>sim
Simulator Version x.x
Enter machinecode file name and/or args, or hit ENTER to quit.
fig0308
Starting session.  Enter h or ? for help.
---- [T7] 0: ld   /0 004/ u*
  0: ld  /0 004/ add  /2 005/ st   /1 006/ halt /FFFF /
  4: ld  /0 00F/ ld   /0 001/ ld   /0 000/
---- [T7] 0: ld   /0 004/      ←hit ENTER to invoke T7
  0: ld   /0 004/ ac=0000/000F
  1: add  /2 005/ ac=000F/0010
  2: st   /1 006/ m[006]=0000/0010
  3: halt /FFFF /
Machine inst count =      4 (hex) =       4 (dec)
---- [T7] q
```

You can also enter the “.mac” file name on the command line when you invoke sim (then sim will not prompt for one):

sim fig0308

Assembler listing (see next slide for an example)

- When **mas** assembles an assembly language program, it also creates a listing file whose extension is “.lst”.
- The listing shows the location and object code for each assembly language statement.
- The listing also provides a symbol/cross-reference table.

FIGURE 3.16 Machine-level Assembler Version x.x

```
  LOC    OBJ    SOURCE
hex*dec

                ; assembly language program that adds two numbers

0  *0    0004             ld    n1       ; get first number
1  *1    2005             add   n2       ; add second number
2  *2    1006             st    result   ; store sum in memory
3  *3    FFFF             halt           ; return to operating system

                ;         data area
4  *4    000F  n1:        dw    15       ; first number
5  *5    0001  n2:        dw    1        ; second number
6  *6    0000  result:    dw    0        ; store sum here
7  *7    ========= end of fig0308.mas ===========================


         Symbol/Cross-Reference Table

Symbol                     Address      References
                            (hex)         (hex)

 n1                           4             0
 n2                           5             1
 result                       6             2

Input file     = fig0308.mas
Output file    = fig0308.mac
List file      = fig0308.lst
Number base   = decimal
Label status  = case sensitive
```

The H1 Software Package has two assemblers: **mas** (the full-featured stand-alone assembler) and the assembler built into the debugger that is invoked with the **a** command.

Assembler built into the debugger

- Labels not allowed unless a special source tracing mode is invoked.
- Comments not allowed
- Blank lines not allowed
- Listing not generated
- Instructions are assembled directly to memory.
- Numbers are hex unless suffixed with “t”

FIGURE 3.17

```
C:\H1>sim
Simulator Version x.x
Enter machinecode filename and/or args, or hit ENTER to quit.
none
Starting session.  Enter h or ? for help.
---- [T7] 0: ld   /0 000/ a0           ←start assembling to loc 0
   0:  ld   /0 000/ ld 4               ←absolute address required
   1:  ld   /0 000/ add 5              ←absolute address required
   2:  ld   /0 000/ st 6               ←absolute address required
   3:  ld   /0 000/ halt
   4:  ld   /0 000/ dw 15t             ←"t" needed to specify decimal
   5:  ld   /0 000/ dw 1
   6:  ld   /0 000/ dw 0
   7:  ld   /0 000/                    ←hit ENTER to exit assembly mode
---- [T7] 0: ld   /0 004/ f 0 6        ←write program in memory to a file
Enter file name.    [f.mac]
simple
Writing locations 0 - 6 to simple.mac
---- [T7] 0: ld   /0 004/ u*           ←unassemble entire program
  0: ld   /0 004/ add  /2 005/ st   /1 006/ halt /FFFF /
  4: ld   /0 00F/ ld   /0 001/ ld   /0 000/
---- [T7] 0: ld   /0 004/              ←hit ENTER to invoke default command
  0: ld   /0 004/ ac=0000/000F
  1: add  /2 005/ ac=000F/0010
  2: st   /1 006/ m[006]=0000/0010
  3: halt /FFFF /
Machine inst count =     4 (hex) =      4 (dec)
---- [T7] q
```

Low-level versus high-level languages

```
w = x + y + z;
```

might be translated to the four-instruction machine instruction sequence corresponding to the assembly instructions

```
ld   x
add  y
add  z
st   w
```

How an assembler works

- It assembles machine instructions using two tables: the *opcode table* and the *symbol table*.
- The opcode table is pre-built into the assembler.
- The assembler builds the symbol table.
- Assembler makes two passes.
- Assembler builds symbol table on pass 1.
- Assembler “assembles” (i.e., constructs) the machine instructions on pass 2.

FIGURE 3.18

Opcode Table
(Part of the Assembler)

Mnemonic	Opcode (hex)
ld	0
st	1
add	2
.	.
.	.
.	

location_counter used to build symbol table

FIGURE 3.19

```
                        ld    x
                        st    y
Pass 1 scan             halt
is here →     x:        dw    5
              y:        dw    0
              z:        dw    x
```

location_counter

$\boxed{3}$

FIGURE 3.20

Symbol Table
(Built by the Assembler)

Symbol	Address (hex)
x	3
y	4
z	5

Assembling the ld x instruction

- Assembler obtains the opcode corresponding to the “ld” mnemonic from the opcode table.
- Assembler obtains the absolute address corresponding to “x” from the symbol table.
- Assembler “assembles” opcode and address into a machine instruction using the appropriate number of bits for each field.

Dup modifier

```
table:   dw   0
         dw   0
         dw   0
         dw   0
         dw   0
```

is equivalent to

```
table:   dw   5 dup 0
```

dup affects location_counter during pass 1

```
table:     dw    7       ; 1st
           dw    7       ; 2nd
            .    .
            .    .
            .    .
           dw    7       ; 1000th
```

When an assembler scans a **dw** directive with a **dup** modifier during pass 1, it must increment **location_counter** to reflect the actual number of words that are defined. For example, suppose **location_counter** contains 50 decimal when the line labeled with **table** in the following sequence is scanned during pass 1:

```
            .
            .
            .
table:     dw    1000 dup 7 ; location_counter = 50
x:         dw    22         ; location_counter = 1050
            .
            .
            .
```

Special forms in operand field

- Label + unsigned_number
- Label – unsigned_number
- *
- * + unsigned_number
- * - unsigned_number

FIGURE 3.21

```
LOC      OBJ   SOURCE
hex*dec

0  *0    0006          ld   table
1  *1    2007          add  table + 1
2  *2    2008          add  table + 2
3  *3    1005          st   table - 1
4  *4    FFFF          halt
5  *5    0000          dw   0
6  *6    0008  table:  dw   8
7  *7    0006          dw   6
8  *8    0004          dw   4
9  *9    ========= end of fig0321.mas ==========================
```

FIGURE 3.22

```
LOC      OBJ    SOURCE
hex*dec

0  *0    0004           ld   x
1  *1    2005           add  y
2  *2    1006           st   * + 4   ; stores into z
3  *3    FFFF           halt
4  *4    0001  x:       dw   1
5  *5    0002  y:       dw   2
6  *6    0000  z:       dw   0
7  *7    ========= end of fig0322.mas ===========================
```

Defining pointers

FIGURE 3.23

```
                ; assume x corresponds to location 50
dw   7          ; 7 is a constant
dw   x          ; points to location 50
dw   x + 2      ; points to location 52
dw   x - 3      ; points to location 47
dw   *          ; points to this location
dw   * - 5      ; points to first dw above
```

ASCII

- Code in which each character is represented by a binary number.
- ‘A’ 01000001
- ‘B’ 01000010
- ‘a’ 01100001
- ‘b’ 01100010
- ‘5’ 00110101
- ‘+’ 00101011

The *null character* is a word (or a byte on a byte-oriented computer) that contains all zeros.

A *null-terminated string* has a null character as its last character.

Double-quoted strings are null terminated:

“hello”

Single-quoted strings are not null terminated:

‘hello’

Double quoted string “ABC” is null terminated

FIGURE 3.25 "ABC" in memory

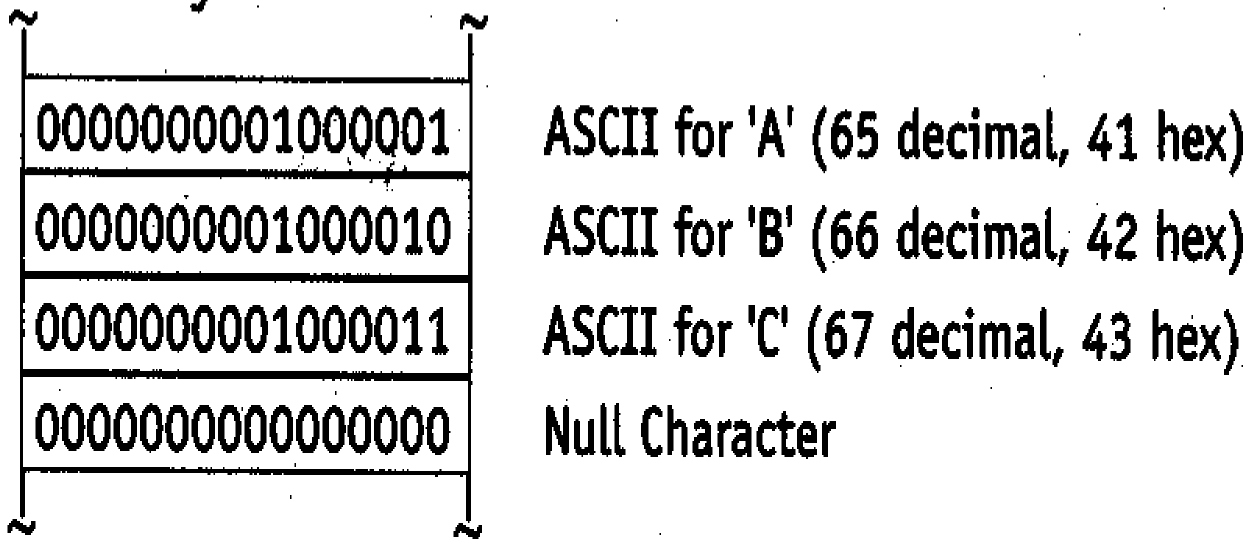
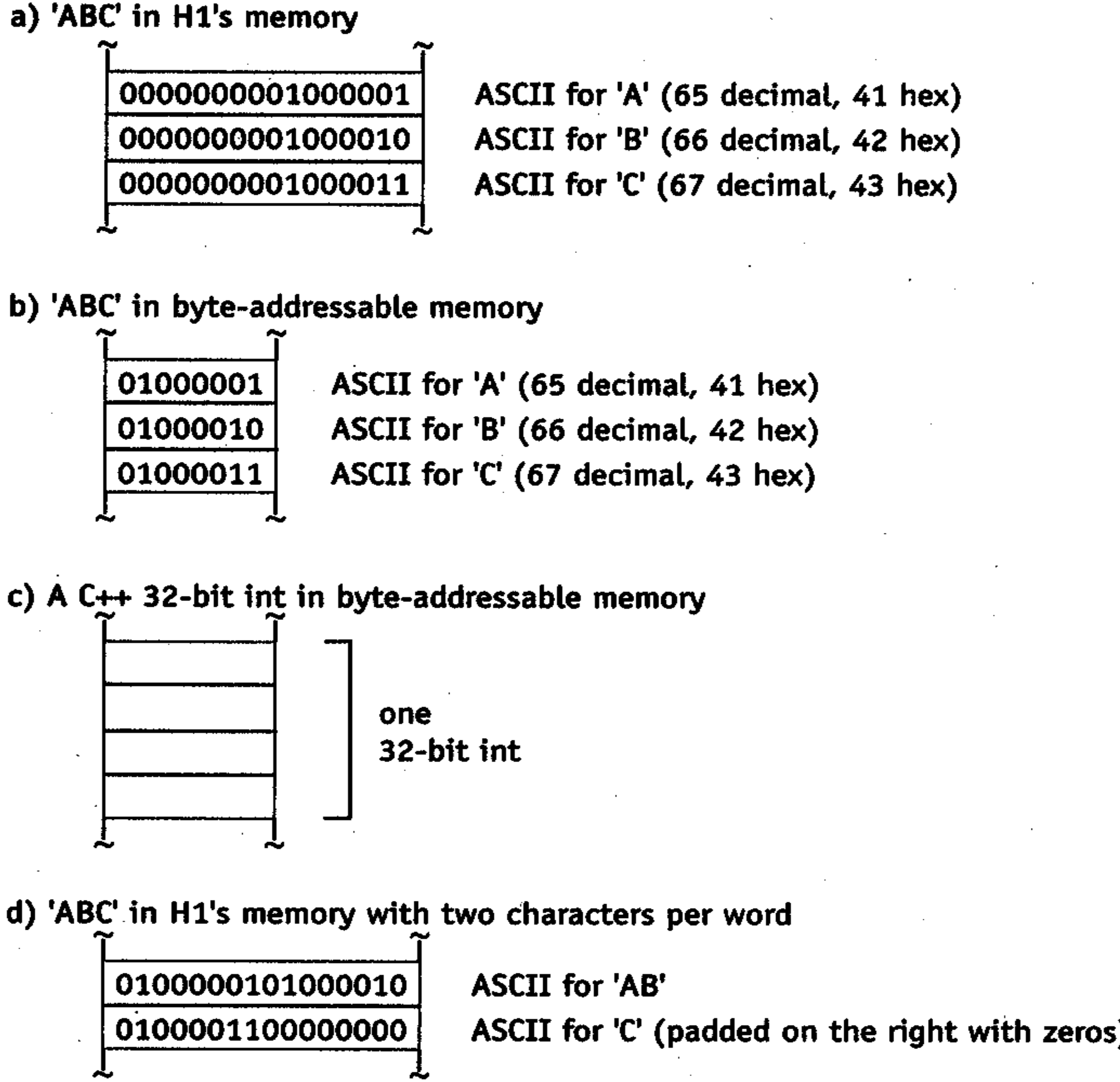
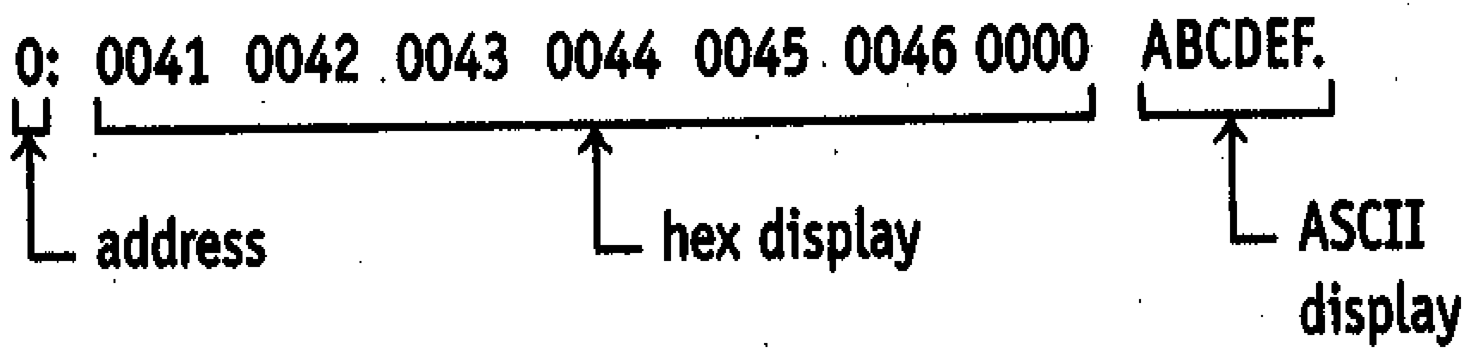


FIGURE 3.24


a) 'ABC' in H1's memory
0000000001000001
0000000001000010
0000000001000011
ASCII for 'A' (65 decimal, 41 hex)
ASCII for 'B' (66 decimal, 42 hex)
ASCII for 'C' (67 decimal, 43 hex)
b) 'ABC' in byte-addressable memory
01000001
01000010
01000011
ASCII for 'A' (65 decimal, 41 hex)
ASCII for 'B' (66 decimal, 42 hex)
ASCII for 'C' (67 decimal, 43 hex)
c) A C++ 32-bit int in byte-addressable memory
one
32-bit int
d) 'ABC' in H1's memory with two characters per word
0100000101000010
0100001100000000
ASCII for 'AB'
ASCII for 'C' (padded on the right with zeros)

```
dw    'ABC'
dw    "DEF"
```

If we display memory with the d* command, we get



An assembly listing shows the object code for only the first occurrence of the data item that follows dup.

See the next slide.

FIGURE 3.26

```
LOC     OBJ   SOURCE
hex*dec

0  *0   0041  s1:      dw    'ABC'
1  *1   0042
2  *2   0043
3  *3   0041  s2:      dw    "ABC"
4  *4   0042
5  *5   0043
6  *6   0000
7  *7   0041  s3:      dw    10 dup "ABC"
8  *8   0042
9  *9   0043
A  *10  0000
2F *47  ========= end of fig0326.mas =============================
```

Escape sequences

FIGURE 3.27

\0	null character
\"	double quote
\'	single quote
\\	backslash
\a	bell
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab

Org directive

- Resets the location_counter to a higher value during the assembly process.
- Reserves but does not initialize an area of memory.

FIGURE 3.28

```
  LOC    OBJ   SOURCE
hex*dec

0   *0    03E8           ld   x               ; load from location 1000
1   *1    1005           st   dataarea        ; store into location 5
2   *2    1006           st   dataarea + 1    ; store into location 6
3   *3    0004           ld   dataarea - 1    ; load from location 4
4   *4    FFFF           halt
              dataarea:  org  1000
3E8*1000 0005 x:         dw   5
3E9*1001 ========= end of fig0328.mas ===========================
```

End directive

- Specifies the *entry point* (i.e., where execution starts) of a program
- If an end directive is omitted, the entry point defaults to the physical beginning of the program.
- And end directive may appear on any line in a program.

FIGURE 3.29

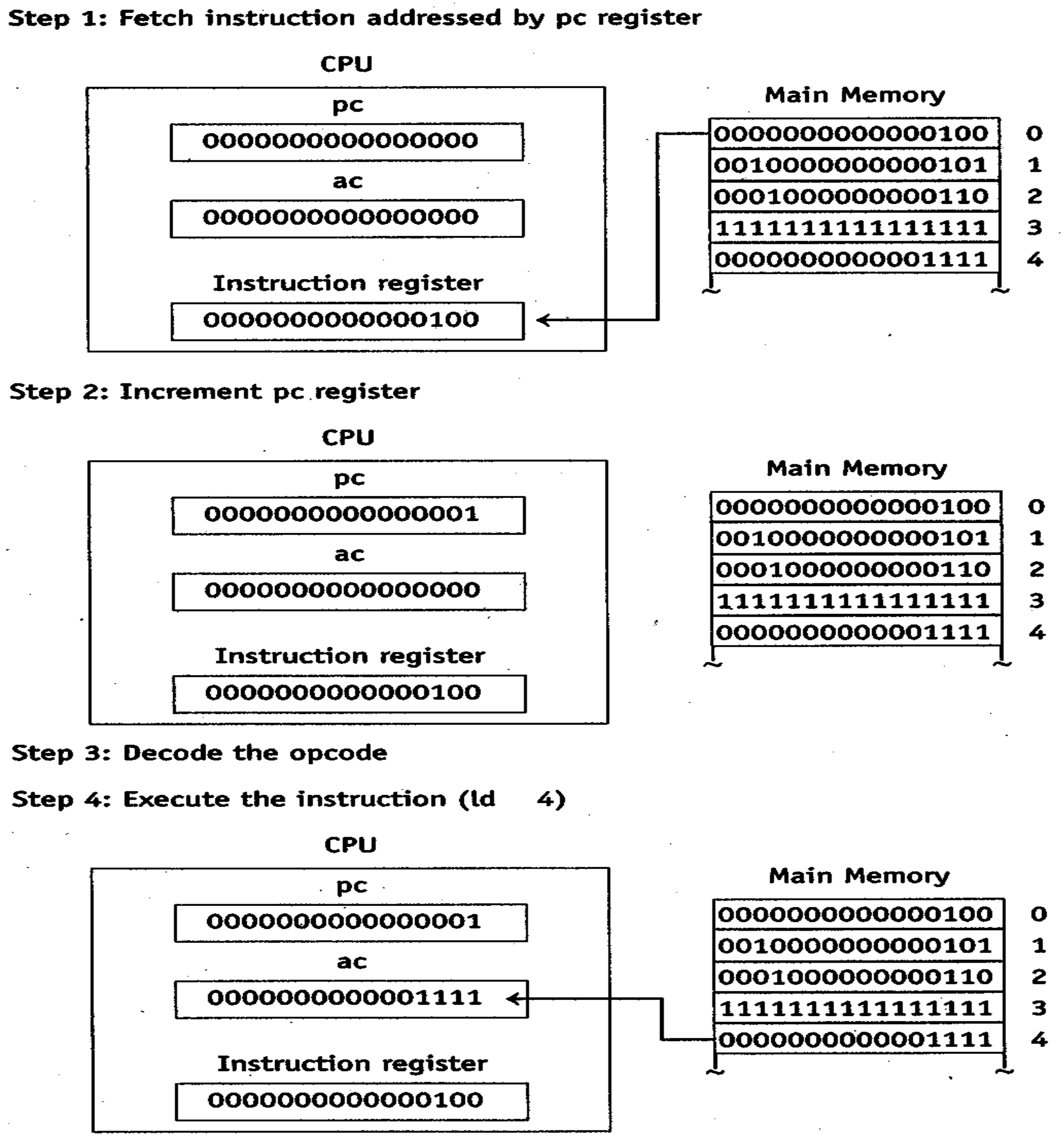
```
LOC      OBJ   SOURCE
hex*dec

0  *0    0001  x:      dw   1
1  *1    000F  y:      dw   15
2  *2    0000  z:      dw   0
3  *3    0000  start:  ld   x      ; execution should start here
4  *4    2001          add  y
5  *5    1002          st   z
6  *6    FFFF          halt
                       end  start
7  *7    ========= end of fig0329.mas ===========================
```

Sequential execution of instructions—the CPU repeatedly performs the following operations:

- Fetch instruction pointed to by pc register
- Increment pc register
- Decode opcode
- Execute instruction

FIGURE 3.30


Step 1: Fetch instruction addressed by pc register
CPU
pc
0000000000000000
ac
0000000000000000
Instruction register
0000000000000100
Main Memory
0000000000000100 0
0010000000000101 1
0001000000000110 2
1111111111111111 3
0000000000001111 4
Step 2: Increment pc register
CPU
pc
0000000000000001
ac
0000000000000000
Instruction register
0000000000000100
Main Memory
0000000000000100 0
0010000000000101 1
0001000000000110 2
1111111111111111 3
0000000000001111 4
Step 3: Decode the opcode
Step 4: Execute the instruction (ld 4)
CPU
pc
0000000000000001
ac
0000000000001111
Instruction register
0000000000000100
Main Memory
0000000000000100 0
0010000000000101 1
0001000000000110 2
1111111111111111 3
0000000000001111 4

Warning

The CPU will fetch and "execute" data

See the next slide.

FIGURE 3.31

```
LOC       OBJ   SOURCE
hex*dec

0  *0     0001           ld   x        ; load -1
1  *1     FFFF  x:       dw   -1
2  *2     2005           add  y        ; add 5
3  *3     1006           st   z        ; store result
4  *4     700B           halt
5  *5     0005  y:       dw   5
6  *6     0000  z:       dw   0
7  *7     ========= end of fig0331.mas ===========================
```

Automatic generation of instructions

- Unlike high-level languages, the assembler does not automatically generate instructions.
- For example, in assembly language you must specify the end-of-module instruction.

FIGURE 3.32

```
LOC     OBJ   SOURCE
hex*dec

0  *0   0005  x:       dw   5
1  *1   0000  epoint:  ld   x          ; need halt instruction
                       end  epoint
2  *2   ========= end of fig0332.mas ============================
```

FIGURE 3.33

```
1 #include <iostream>
2 using namespace std;
3 void main()
4 {
5    cout << "hello\n";
6 }
```